Proseminar Computergraphik
Rastergraphik-Algorithmen

Patrick Bergner

20. Juni 2006

Institut für Software- und Multimediatechnik
Fakultät Informatik
Technische Universität Dresden

Zusammenfassung

Das vorliegende Dokument ist aus einem Vortrag im Rahmen des Proseminar Computergraphik im Sommersemester 2006 an der Technischen Universität Dresden entstanden.

Es behandelt grundlegende Aspekte der Themen Rasterkonvertierung von Linien und Kurven, Füllen von Polygonen und geschlossenen Kurven sowie Clippen von Linien und Polygonen an Polygonen.

Zudem befinden sich im Anhang ausgewählte Implementationsbeispiele aufgeführter Algorithmen.

1. Einleitung

Zunächst soll erläutert werden, was eine Rastergraphik ist und durch welche Merkmale sie gekennzeichnet ist.

Eine Rastergraphik (engl. Bitmap) ist eine matrixförmige Anordnung von Pixeln, denen jeweils eine Farbigkeit und eventuell ein Opazitätswert zugeordnet sind.

Sie wird charakterisiert durch ihre Breite und Höhe in Pixeln sowie ihre Farbtiefe, welche die Anzahl der möglichen Farben für jedes Pixel angibt.

Als Dateiformat werden Rastergraphiken zur Repräsentation von komplexen Bildern wie zum Beispiel Fotos genutzt. Darüberhinaus finden sie Verwendung bei der Ansteuerung von Computerbildschirmen. Diese werden mit einer Rastergraphik aus dem Framebuffer der Grafikkarte angesteuert.

2. Rasterkonvertierung

Der Zweck und gleichzeitig das Problem der Rasterkonvertierung ist die Darstellung stetiger graphischer Objekte durch diskrete Pixel in einem festen Raster.

Dieses Raster ist durch die Auflösung der Rastergraphik definiert. Das Ziel dabei ist eine so genaue aber auch gleichzeitig effiziente Darstellung wie möglich.

Folgende Effekte können bei der Rasterkonvertierung auftreten (Auszug):

- ungleiche Intensität: Linien auf Diagonalen erscheinen schwächer, da dort der relative Pixelabstand größer ist.
- Aliasing: ein Punkt wird nicht an seinem wahren Ort, sondern an einem angenähertem Rasterpunkt (Aliaspunkt) gezeichnet.
- Overstrike: durch die Rasterung mehrerer graphischer Objekte wird ein Rasterpunkt mehrfach gesetzt und gehört damit geometrisch zu mehreren Objekten. Die genaue Zuordnung von Kantenpixeln wird dadurch erschwert, ist aber für Verfahren wie das Füllen oder Clippen essentiell.
- Treppchenbildung: wird der Zoomfaktor auf ein Raster oder das Raster selbst zu groß gewählt, erscheinen Kurven treppchenförmig. Dies kann durch geeignete Verfahren ausgeglichen, jedoch in seiner Ursache nicht eliminiert werden.

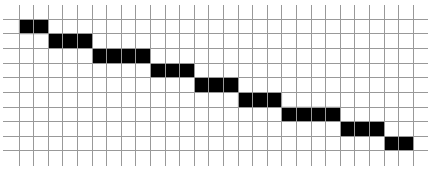

Abbildung 1: Treppchenbildung

2.1. Rasterung von Linien

Zur Rasterung einer einfachen Linie soll zunächst eine intuitive Lösung betrachtet werden.

Dabei wird zu jedem x-Wert der dazugehörige y-Wert mittels Geradengleichung $y = m \times x + n$ bestimmt und entsprechend gerundet. Das Problem dieser Vorgehensweise ist schnell ersichtlich: es wird für jeden Wert eine Gleitkomma-Multiplikation und eine Gleitkomma-Rundungsoperation benötigt. Dies resultiert in unnötigem Zeitaufwand.

Unter der Beobachtung, dass sich der x-Wert bei der Berechnung mit jedem Schritt genau um 1 ändert (äquidistant, treibende Achse), lässt sich folgern, dass sich der y-Wert dabei immer genau um m ändert. In Folge dessen ist nur noch eine Gleitkomma-Addition, statt der Gleitkomma-Multiplikation von Nöten.

Darüberhinaus kann die Rundungsoperation durch Einführung einer Fehlervariablen *error* eingespart werden. Diese zeigt an, um wie viel das zuletzt gesetzte Pixel (x, y) von der idealen Linie abweicht. Wenn $error \leq 0,5$ ist, liegt (x_{+1}, y) näher an der Ideallinie, sonst (x_{+1}, y_{+1}).

Dieses Verfahren ist ohne Erweiterung nur für Linien im ersten Oktanden des kartesischen Koordinatensystems geeignet, da sich der x-Wert immer um 1 und der y-Wert minimal um 0 und maximal um 1 (entspricht 0 bis 45° Steigung) erhöht. Diese Lösung wird als inkrementeller Algorithmus bezeichnet.

Ein sehr viel effektiverer Algorithmus wurde 1962 bei IBM zur Ansteuerung von Plottern von Jack Bresenham entwickelt und trägt auch dessen Namen: Bresenham-Algorithmus.

Er zeichnet sich durch hohe Geschwindigkeit und leichte hardwareseitige Implementierbarkeit aus. Ausgehend vom inkrementellen Algorithmus kann beobachtet werden, dass m nur zur Berechnung von *error* verwendet wird. Dadurch können m und *error* so skaliert werden, dass sie stets ganzzahlige Werte annehmen.

Ohne weiterführende Betrachtung ist der Algorithmus wiederum nur für Linien im ersten Oktand geeignet. Abbildung 2 zeigt eine Übersicht der Oktanden des kartesischen Koordinatensystems.

Zur Berechnung und Implementierung des Algorithmus kann ein rekursives Formelsystem mit einer Entscheidungsvariablen d genutzt werden:

gegeben: zwei Punkte (x_1, y_1) und (x_2, y_2)

$$\begin{aligned} \Delta x &= x_2 - x_1 \\ \Delta y &= y_2 - y_1 \\ d_0 &= 2\Delta y - \Delta x \\ d_1 &= 2\Delta y \\ d_2 &= 2\left(\Delta y - \Delta x\right) \\ d_{i+1} &= d_i + d_2 \text{ für } d_i \geq 0 \\ d_{i+1} &= d_i + d_1 \text{ für } d_i < 0 \end{aligned}$$

2.2. Rasterung von Kurven

Als Beispiel für die Rasterung einer einfachen Kurve zweiter Ordnung soll die Rasterung ganzzahliger Kreise genutzt werden.

Hierzu verwendet man eine abgewandelte Form des Bresenham-Algorithmus zur Rasterung von Linien, welcher diesem allerdings sehr ähnlich ist.

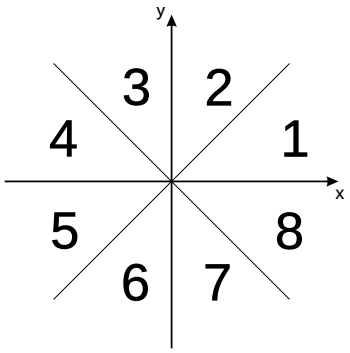

Abbildung 2: Übersicht Oktanden

Der Kreis wird dabei nur im zweiten Oktanden wirklich berechnet, alle anderen Punkte werden durch Spiegelung an den Koordinatenachsen und den Geraden $y = x$ und $y = -x$ hinzugefügt.

Es kann wieder ein Formelsystem zur Berechnung aufgestellt werden:

gegeben: Mittelpunkt (x_1, y_1) und Radius r

$$\begin{aligned} d_1 &= 1 - r \\ d_{i+1} &= d_i + 2x_i + 3 \text{ für } d_i < 0 \\ d_{i+1} &= d_i + 2x_i - 2y_i + 5 \text{ für } d_i \geq 0 \end{aligned}$$

Dabei wird wie folgt vorgegangen:

- x startet bei 0
- y startet bei r
- wenn $d_{i-1} > 0$ ist, wird y_i dekrementiert
- der Algorithmus terminiert, wenn $x > y$ ist, da an dieser Stelle der zweite Oktand verlassen wird

3. Füllen

Zum Füllen von Polygonen und geschlossenen Kurven muss zunächst eine Annahme getroffen werden: der zu füllende Bereich muss entweder durch seine geometrische Beschreibung, zum Beispiel durch eine Liste von Kanten-Endpunkten oder durch Begrenzungspixel auf dem Bildschirm, bzw. im Framebuffer, gegeben sein.

3.1. Scangeraden-Methode

Eine Möglichkeit zum Füllen von Flächen ist die Scangeraden-Methode.

Diese setzt die geometrische Beschreibung des zu füllenden Polygons voraus. Es wird parallel zur x-Achse eine Scangerade von unten nach oben über das zu füllende Polygon gezogen und dabei die Schnittpunkte S zwischen Scangerade und Polygon berechnet.

Für konkave Polygone ergibt sich dabei immer eine gerade Anzahl von Schnittpunkten.

Daraus folgt, dass im Allgemeinen für konkave Polygone immer die Segmente zwischen S_{2i-1} und S_{2i} für $1 \leq i \leq n$ eingefärbt werden müssen.

Vorsicht ist beim Füllen von Polygonen geboten, die über bestimmte Bereichsgrenzen, zum Beispiel die Bildschirmkanten, hinausragen und abgeschnitten werden.

Der Scangeraden-Algorithmus kann dabei eventuell nicht korrekt bestimmen, welcher Bereich innerhalb des Polygons liegt und welcher nicht. Daher müssen in solchen Fällen die Bereichsgrenzen als zusätzliche Polygonkanten übergeben werden.

Der Algorithmus arbeitet nach folgendem Prinzip:

- eine äußere Schleife läuft über die Kanten des Polygons
- eine innere Schleife berechnet alle Schnittpunkte einer Polygonkante mit allen Scangeraden zwischen y_{min} und y_{max}
- die gefundenen Schnittpunkte werden in einer geeigneten Datenstruktur abgelegt und nach der Abarbeitung des gesamten Polygons ausgewertet
- die gefundenen Linien werden zum Beispiel mit dem Bresenham-Algorithmus zur Rasterung von Linien gefärbt

3.2. Saatfüllen

Eine weitere Möglichkeit zum Füllen von Polygonen ist das Saatfüllen (engl. seedfill oder floodfill). Dieser Algorithmus benötigt die implizite Darstellung eines Polygons durch Kantenpixel in einer definierten Grenzfarbe.

Die Grundidee ist denkbar einfach: ausgehend von einem Pixel im zu füllenden Bereich werden rekursiv alle anderen gefüllt, bis man die Kantenpixel erreicht.

Beachtet werden muss dabei allerdings, dass keine rekursiven Aufrufe für diagonal benachbarte Pixel ausgeführt werden. Da durch die Rasterkonvertierung von Polygonen Treppchenbildung auftritt, könnte ein diagonaler Aufruf aus dem Polygon hinaus führen. Folglich würden mehr Bereiche gefüllt, als eigentlich gewünscht.

Ein großer Nachteil des Saatfüllens ist der enorm hohe Aufwand für den Rekursionskeller. In Folge dessen sind Pufferüberläufe nicht nur möglich, sondern sehr wahrscheinlich. In einem begrenzten Maß schafft die Nutzung von Pixelläufen dabei Abhilfe.

Ein Pixellauf umfasst eine Gruppe von horizontal benachbarten Pixeln, die auf beiden Seiten von Pixeln der Grenzfarbe eingeschlossen sind und dabei kein Pixel der Grenzfarbe enthalten. Pixelläufe werden an Hand ihres rechtesten Pixels erkannt und verarbeitet, wobei dies eine rein willkürliche Festlegung ist.

Zu Beginn wird der Pixellauf ermittelt, der das Startpixel enthält. Indem man vom Startpixel aus soweit nach rechts läuft, bis man die Kantenpixel erreicht, erhält man den ersten Pixellauf. Das letzte Pixel, welches nicht zur Polygonkante gehört, wird als erster Pixellauf gespeichert.

Von diesem Pixellauf aus wird in den nach oben und unten benachbarten Zeilen ebenfalls nach Pixelläufen gesucht und diese gegebenenfalls auf einem Kellerspeicher abgelegt.

Der Rekursionsaufwand wird verringert, da dabei nur vertikale Aufrufe erfolgen. Nach der Untersuchung des Polygons werden alle Pixelläufe mit der Füllfarbe gefüllt.

4. Clipping

Als Clippen wird das Abschneiden von Polygonen bezüglich einem anderen Polygon (Clip-Polygon) bezeichnet. Das Clippen von Geradensegmenten bezüglich einem rechteckigen Clip-Polygon ist eine der häufigsten und grundlegendsten Operationen der Computergraphik.

Es gibt zwei Möglichkeiten Clipping durchzuführen. Eine ist die Schnittpunkte des zu clippenden Objektes mit dem Clip-Polygon zu berechnen und anschließend mit Hilfe dieser Daten zu clippen. Eine weitere ist das Clip-Polygon durch die Auflösung des Bildschirms als Rechteck zu definieren und direkt während der Bildschirmdarstellung zu clippen.

4.1. Clippen von Linien

Das Clippen von Linien bezüglich einem rechteckigem Clip-Polygon wird am häufigsten verwendet, um Objekte zu clippen, die über die Grenzen eines Bildschirmbereichs herausragen.

Zunächst soll eine analytische Möglichkeit betrachtet werden. Sei dafür S ein Geradensegment mit den dazugehörigen Endpunkten. Liegen beide Endpunkte innerhalb des Clip-Polygons, liegt ganz S innerhalb des Clip-Polygons und es muss kein Teil von S abgeschnitten werden.

Liegen beide Endpunkte außerhalb des Clip-Polygons, kann S teilweise oder ganz außerhalb des Clip-Polygons liegen. Um dies herauszufinden sind alle Schnittpunkte zwischen S und dem Clip-Polygon zu bestimmen.

Werden keine Schnittpunkte gefunden, liegt S ganz außerhalb des Clip-Polygons. Ansonsten wird eine Gerade zwischen den gefundenen Schnittpunkten gezeichnet und das Clipping ist abgeschlossen.

Einen algorithmischen Ansatz und gleichzeitig einen Versuch der Vermeidung von unnötigen Schnittpunktberechnungen stellt der Algorithmus von Cohen und Sutherland dar.

Dieser arbeitet mit einem Modell, welches das Gebiet um das Clip-Polygon in neun Bereiche unterteilt und jedem Bereich einen 4 Bit-Code zuordnet. Abbildung 3 zeigt dieses Modell. Die einzelnen Bits sind dabei wie folgt gesetzt:

- Bit 3 für Bereiche oberhalb des Clip-Polygons
- Bit 2 für Bereiche unterhalb des Clip-Polygons
- Bit 1 für Bereiche rechts des Clip-Polygons
- Bit 0 für Bereiche links des Clip-Polygons

Die Codes von benachbarten Bereichen unterscheiden sich jeweils in nur einer Bitstelle. Der Bereich mit dem Code 0000 genau in der Mitte stellt den sichtbaren Bildschirmbereich dar.

1001	1000	1010
0001	0000	0010
0101	0100	0110

Abbildung 3: Bereichscodes

Für beliebige Endpunkte (x, y) eines Geradensegmentes S ergeben sich die Bereichscodes durch:

- Bit 3 ist das Vorzeichenbit der Operation $y_{max} - y$
- Bit 2 ist das Vorzeichenbit der Operation $y - y_{min}$
- Bit 1 ist das Vorzeichenbit der Operation $x_{max} - x$
- Bit 0 ist das Vorzeichenbit der Operation $x - x_{min}$

Dabei sind (x_{min}, y_{min}), (x_{max}, y_{min}), (x_{min}, y_{max}) und (x_{max}, y_{max}) die Eckpunkte des rechteckigen Clip-Polygons.

Wenn die bitweise ODER-Verknüpfung zweier Bereichscodes 0000 ergibt, liegt das gesamte Geradensegment S innerhalb des Clip-Polygons.

Wenn die bitweise UND-Verknüpfung zweier Bereichscodes ungleich 0000 ist und die beiden Bereichscodes an derselben Stelle eine 1 besitzen, liegt das gesamte Geradensegment S außerhalb des Clip-Polygons.

Erst wenn diese beiden einfachen Tests kein eindeutiges Ergebnis gezeigt haben, wird das Geradensegment S an einer Kante des Clip-Polygons in zwei Teile geclippt und die beiden Tests wiederholt. Dies geschieht solange, bis alle Geradensegmente einen der beiden Tests erfüllen und damit vollständig und korrekt geclippt sind.

4.2. Clippen von Polygonen

Exemplarisch für das Clippen von Polygonen bezüglich einem rechteckigen Clip-Polygon soll die Arbeitsweise des Algorithmus von Sutherland und Hodgman gezeigt werden.

Der Algorithmus arbeitet nach dem "Divide and Conquer"-Prinzip. Das bedeutet er löst eine Reihe von einfachen Problemen, deren Lösungen zusammen die Lösung für das Gesamtproblem ergeben. Das einfache Problem in diesem Fall ist das Clippen eines Polygons an einer einzelnen unendlichen Geraden.

Im Unterschied zum Algorithmus von Cohen und Sutherland clippt der Algorithmus von Sutherland und Hodgman das Polygon nacheinander an allen Kanten des Clip-Polygons, während der Algorithmus von Cohen und Sutherland zunächst berechnet, ob überhaupt geclippt werden muss und clippt nur bei Bedarf.

Dieses Verfahren ist deutlich aufwendiger, allerdings kann dadurch jedes konkave oder konvexe Polygon an jedem konvexen Polygon geclippt werden.

Die Eingabe für den Algorithmus stellen dabei die Koordinaten der Polygonknoten dar. Bei Aufruf des Algorithmus clippt dieser das Polygon an einer einzelnen unendlichen Kante des Clip-Polygons und gibt wieder eine Anzahl von Knoten aus, die das geclippte Polygon beschreiben.

Bei Ausgabe eines Knotens wird der Algorithmus rekursiv mit dem neuen Knoten aufgerufen und durchläuft den Algorithmus nochmals.

Das Ergebnis ist, dass das Polygon eine Pipeline von Clipping-Funktionen durchläuft, ohne dass es dabei zwischengespeichert werden muss.

Dies macht den Algorithmus enorm leistungsfähig, da er wiedereintrittsfähig wird und in Hardware ohne Pufferspeicher implementiert werden kann.

Darüberhinaus kann er nicht nur in der 2D-Computergraphik, sondern auch zum Clippen von dreidimensionalen Objekten genutzt werden.

5. Quellen

[1] Dr. rer. nat. Thomas Rauber: „Algorithmen in der Computergraphik“, 1993

[2] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, Richard L. Phillips: „Grundlagen der Computergraphik. Einführung, Konzepte, Methoden.“, 1994

[3] Prof. Dr.-Ing. F. Jaeger: Vorlesungsskript Computergrafik, HTWK Leipzig, 2005

A. Beispielimplementationen

A.1. inkrementeller Algorithmus zur Rasterung einer Linie

```
void line (int x1, int y1, int xn, int yn) {

  float m, error;
  int x, y;

  m = ((float) (yn - y1)) / (xn - x1);
  y = y1;
  error = 0.0;

  for (x = x1; x <= xn; x++) {

    setPixel(x, y);
    error += m;

    if (error >= 0.5) {

      y++; error--;

    }
  }
}
```

A.2. Bresenham-Algorithmus zur Rasterung einer Linie

```
void bresenham (int x1, int y1, int xn, int yn) {

  int error, x, y, dx, dy;

  dx = xn - x1;
  dy = yn - y1;
  error = -dx/2;
  y = y1;

  for (x = x1; x <= xn; x++) {

    setPixel(x, y);
    error += dy;
    if (error >= 0) {

      y++;
      error -= dx;

    }
  }
}
```

A.3. Bresenham-Algorithmus zur Rasterung eines ganzzahligen Kreises

```
void bresenhamkreis(int x1, int y1, int radius) {

    int f = 1 - radius;
    int x = 0;
    int y = radius;
    int dx = 0;
    int dy = -2 * radius;

    setPixel(x1, y1 + radius);
    setPixel(x1, y1 - radius);
    setPixel(x1 + radius, y1);
    setPixel(x1 - radius, y1);

    while(x < y) {

        if(f >= 0) {

          y--;
          dy += 2; f += dy;

        }

        x++;
        dx += 2;
        f += dx + 1;

        setPixel(x1 + x, y1 + y);
        setPixel(x1 - x, y1 + y);
        setPixel(x1 + x, y1 - y);
        setPixel(x1 - x, y1 - y);
        setPixel(x1 + y, y1 + x);
        setPixel(x1 - y, y1 + x);
        setPixel(x1 + y, y1 - x);
        setPixel(x1 - y, y1 - x);

    }
}
```

A.4. Saatfüllen ohne Pixelläufe

```
void seedfill(int x, int y, int bcol, int fillcol) {

  if ((pixel_value(x,y) != bcol) && (pixel_value(x,y) != fillcol)) {

      setPixel(x, y, fillcol);

      seedfill(x+1, y, bcol, fillcol);
      seedfill(x-1, y, bcol, fillcol);
      seedfill(x, y+1, bcol, fillcol);
      seedfill(x, y-1, bcol, fillcol);

  }
}
```

A.5. Algorithmus von Cohen und Sutherland zum Clippen von Linien

```
void clip (float x1, float y1, float x2, float y2,
  float xmin, float xmax, float ymin, float ymax) {

    int c1, c2;
    float xs, ys;

    c1 = code(x1,y1);
    c2 = code(x2,y2);

    if (c1 | c2 == 0x0) line(x1, y1, x2, y2);
    else if (c1 & c2 != 0x0) return;

  else {

    intersect(xs, ys, x1, y1, x2, y2, xmin, xmax, ymin, ymax);

    if ( is_outside(x1,y1))
      clip(xs, ys, x2, y2, xmin, xmax, ymin, ymax);

    else
      clip(x1, y1, xs, ys, xmin, xmax, ymin, ymax);

  }
}
```